http://bahramelectronic.tk

Microcontroller Basics Course

part 1: the TASM assembler

bahramelectronic.tk

By B. Kainka

This course is for everyone who always wanted to how microcontrollers work and how to use them, but was always afraid to ask. It is intended to explain the fundamentals, starting from scratch. The Elektor Electronics 89S8252 Flash Microcontroller Board (presented in last month's issue) is used as the hardware platform.

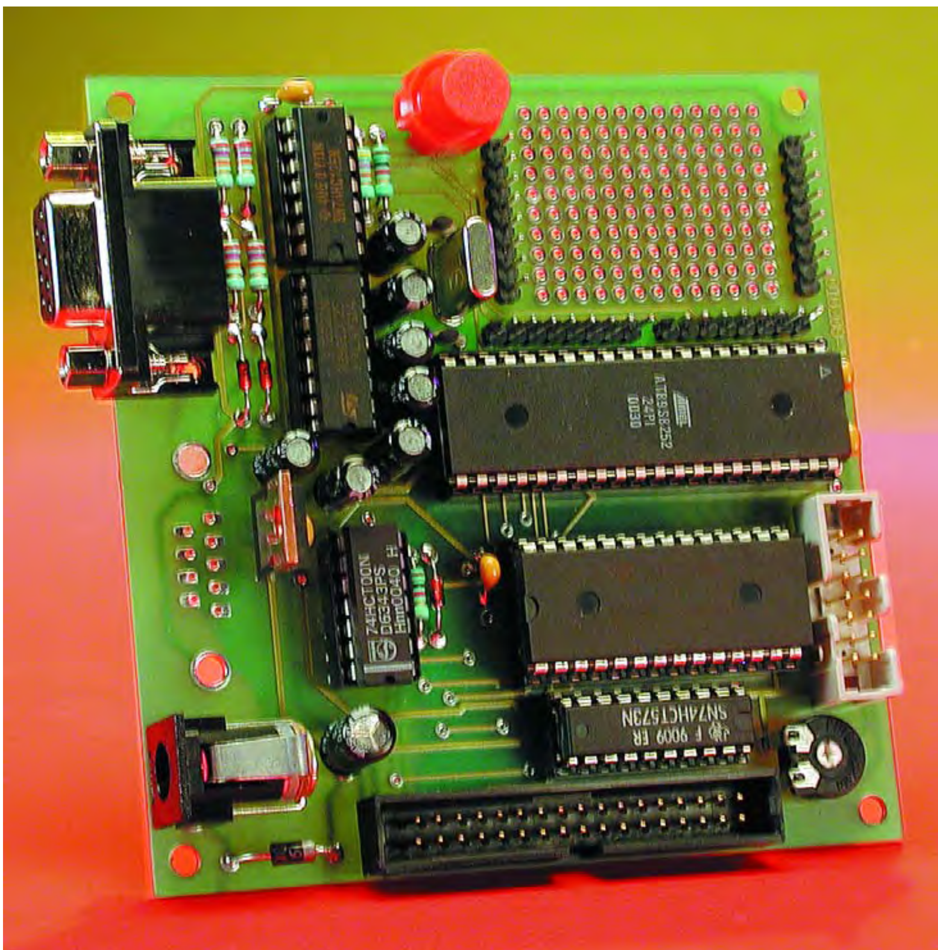

Figure 1. The 89S8252 Flash Board, which is used in this course, is a general-purpose microcontroller system.

bahramelectronic@bahramelectronic.tk

Nowadays, we all take working with computers for granted, and not only that, we often work with quite powerful equipment. The heart of a computer is its processor, such as a Pentium III. Relatively speaking, a microcontroller is both much less and much more than the processor of a typical PC. It is less because it processes smaller programs, uses less memory and is usually much slower. However, it is also more because it already has many elements on a single chip that are spread out over the complete motherboard of a PC, namely working memory, timers, interfaces and port connections. What makes microcontrollers attractive is that is in the limiting case, a complex problem can be solved using only a single IC. Using programming alone, anyone can produce a special IC that does exactly what he or she wants — and at a relatively low cost.

A microcontroller is thus some-

http://bahramelectronic.tk

thing like a logical circuit with many possible inputs and outputs. What this circuit does is determined by a program. Perhaps you want to build a digital counter, or would you rather have a stopwatch? Do you want to create a special logical gate, or perhaps a universal clock generator? Do you need to decode a complicated digital signal or control a digital circuit? In all of these cases, a microcontroller can help you. There are many examples of problems whose solutions previously required an enormous board full of ICs and now can be solved quite elegantly by a single IC, namely a microcontroller. Consequently, some knowledge of programming is worth having. There are many different approaches that can be taken to achieve this goal.

The hardware basis for this course is the **89S8252 Flash Microcontroller Board** described in last month's issue of *Elektor Electronics* (see **Figure 1**). As already announced, for programming software we will use the following three programming languages: assembler, Basic and C. Our first experiments will be carried out in assembler. Why should we use assembler in particular? Isn't it rather difficult, perhaps too difficult for beginners? The answer is no, since the initial examples will be very small and easy to understand. The advantage or using assembler is that it allows us to work very close to the hardware, so we can see exactly what is happening. High-level languages (such as BASIC), by contrast, hide much of what actually takes place.

In our first experiment, all we want to do is to switch the levels on one of the microcontroller's output ports. After all, operating a switch is the first step in automation. Also, the results can be observed using a voltmeter connected to lead P1.4 via connector K4 (see **Figure 2**).

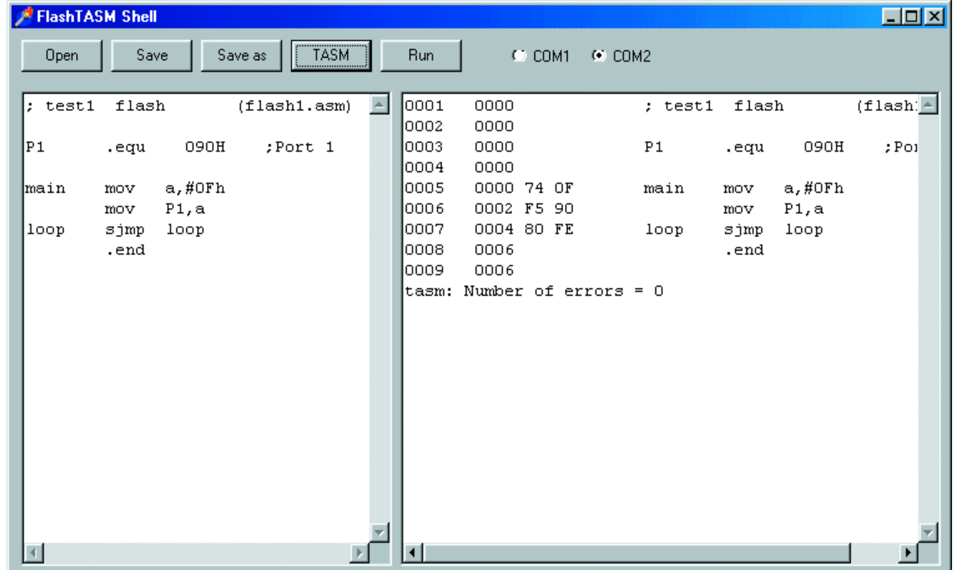

Figure 3. The first sample program in TASMedit.exe.

In order to change the level on this lead, we will use a small assembler program. Put briefly, assembler is a notation used to write instructions for a processor or microcontroller. Every microcontroller has an instruction set, which ultimately consists of numerical values and associated functions. The following series of six numbers represents a small, complete program for an 89S8252 microcontroller — in fact, it is what is called a machine-language program:

```
116,15,245,144,128,254
```

It is generally customary to write computer programs using hexadecimal numbers instead of decimal numbers, since the former are easier to read. In hexadecimal notation, the above program looks like this:

```
74 0F F5 90 80 FE
```

We have to write this sequence of numbers into the microcontroller's program memory. We can use a program called MicroFlash for this purpose. Ultimately, the program numbers end up in the program memory of the microcontroller. The microcontroller reads the numbers from the memory, one after the other, and it then knows what it has to do. In normal language, we can express this as follows:

74: So, I'm supposed to transfer a numerical value to the accumulator (that's my memory) — but which one?
0F: Here it is: OF — good, I've made a note of it.
F5: OK, now I have to write the value to a register — but can you please tell me which one?
90: I see, the register for Port 1 is located at address 90. There you are.
80: And now I have to make a short jump — but to where?
FE: Two bytes back from the location that would have been the next one. OK, I'm jumping!
80: The same jump again — OK, I guess I'll just have to keep on running around in a circle.

This is how the microcontroller 'thinks' and acts, since clever engineers have trained it to behave this way. Ultimately, a microcontroller is nothing more than a very complex circuit made up of logic gates. This circuit responds to the states of its input lines, which in this case are the data lines connecting the program memory to the central processing unit.

If we wanted to know what goes on inside a microprocessor in detail, we would have a lot of work on our hands. However, it is sufficient for us to know the machine instructions of a processor and be able to use them. This means that the microcontroller itself remains a sort of 'black box', whose inner functions

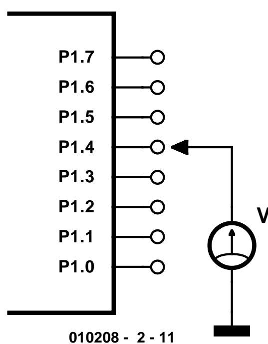

Figure 2. The results of the first experiment can be checked using a voltmeter.

bahramelectronic@bahramelectronic.tk

http://bahramelectronic.tk

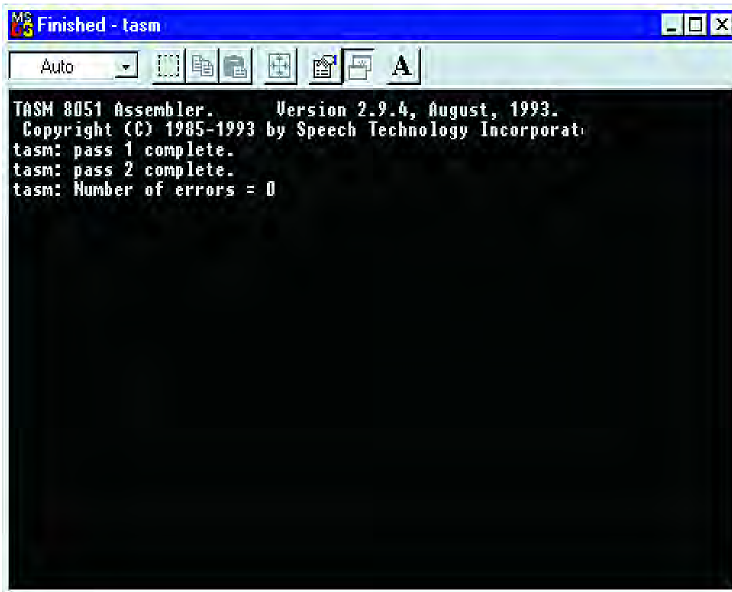

Figure 4. TASM in the DOS window.

we do not fully know but whose behaviour is easy to understand. That's how it is with modern technology — it has become nearly impossible to regard everything at all possible levels of understanding.

So, now we know that a microcontroller has its own language, which actually consists of nothing but numbers. However, there is an obvious problem: the language that a microcontroller can read easily and fluently is not exactly suitable for people. We are not made to work with numbers, but rather with words. Consequently, words (which are easier to remember) have been devised to represent the individual machine-language instructions. The programmer writes these words in a text file, and a special program then translates them into the language of the microcontroller. This program is called an assembler, and the programming language is also called assembler (or assembly language). Assembler is thus a notation that you and I can use to tell a microcontroller what it should do, as in:

```
main    mov   a,#0Fh
        mov   090H,a
loop    sjmp  loop
```

This is already much more readable. Actually, here we only need to know two special words: 'mov' and 'sjmp'. Both of these words are called mnemonics, which means markers used in place of the actual machine-language instructions. The word 'mov' (move) means 'move', 'shift' or 'load'. Following it comes first the location where something is to be loaded and then what is to be loaded. In the first line, the numerical value 0Fh (= 15), which is identified by the '#' symbol, is loaded into the accumulator a. The accumulator is a register or memory with a size of eight bits, so it can hold numerical values between 0 and 255.

In the second line, the value in the accumulator is then copied to address 90h (= 144). At this address there is a register whose leads are routed to the exterior of the IC, namely to the Port 1 pins. The word 'sjmp' (short jump) causes a jump in program execution, in this case to the location 'loop'. The word 'loop' has been chosen completely arbitrarily and simply represents an address, in this case a position in the series of instructions. The assembler treats such words, which are called labels, as addresses and replaces them with the appropriate numerical values. The sjmp instruction can cause a jump of up to 127 bytes backwards or 128 bytes forwards. A single byte is thus sufficient to specify the jump destination. Here the jump is calculated relative to the current position in the program.

The word 'main' at the beginning of the program is also arbitrarily chosen. The only actual assembler keywords here are thus 'mov' and 'sjmp'. We humans can easily remember such words without the aid of an electronic brain.

However, there is still a problem. Is it really necessary for us to remember that a particular register for Port 1 is located at position 90h? After all, it would be nicer if we could also write this as text. This is easily done; we simply have to define a certain bit of text as being equivalent to a numerical value. The assembler will then replace this text with the corresponding numerical value at every location where the text is found. To make such a definition, we use the assembler directive '.equ' (equate). An assembler directive always begins with a full stop, which informs the assembler that it is not an assembly-language instruction. In the following example, the word 'P1' is assigned the value 90h. In the actual program, the value 90h is thus replaced by 'P1':

```
; flash1.asm  port output

P1    .equ  090H   ;Port 1

main  mov   a,#0Fh
      mov   P1,a
loop  sjmp  loop
     .end
```

This listing also shows us something else: the jump labels and newly defined words are all located at the beginning of the line, and all assembler instructions are located somewhat indented. Furthermore, there is also a comment, which plays absolutely no part in the translation. A comment starts with a semicolon (;).

The specific notation varies somewhat from one assembler to the next. Here we are using the shareware assembler **TASM** (Table-Driven Assembler, a program written by Thomas N. Anderson). TASM is very simple and can translate programs for many different types of microcontrollers, as long as it has the appropriate instruction table.

At the end of the program there is a loop, in which a jump to the destination 'loop' takes place, always and forever. In other programming languages, such a situation would be called a fatal endless loop, which is practically equivalent to a crash. In such a loop, the processor is in a state that it cannot exit under its own power. In general, it should always be clear what should be

bahramelectronic@bahramelectronic.tk

done once the current task has been completed. However, in this case this loop is very important. There is only one task, namely changing the state of the port. If we were to leave the processor to its own devices, it would execute commands that just happen to be in the program memory and perhaps belong to a completely different program. Consequently, a limit must be set by means of an endless loop: this far and no further! In fact, the processor is trapped in this loop, with the only means of escape being a reset. After that, the same program could be started again, or we could load a new program and then run it. Loading a program also takes place in the reset state; the program memory is thus not filled by the processor itself, but by special functional blocks in the microcontroller that program the flash ROM. Each time the board is switched on, a short reset is automatically executed. Following this, the microcontroller finds the most recently loaded program and runs it.

Using the TASM assembler

Now it's time to get down to business! What we want to do is to write this first program, translate it and send it to the microcontroller. For this, we need some software. We will use the well-known shareware assembler TASM, which is located on the working diskette for the course in the form of a zip file; it can also be downloaded from the *Elektor Electronics* website. The file TASM.ZIP must be unpacked into a working directory on the hard disk that will contain the program **TASMedit** and the sample programs.

The special feature of TASM is that is can be used for different types of microcontrollers. For each type there is a table of available machine instructions, which must be identified when the program is started. This is done using a command line; in this case we use

```
TASM -51 -b flash1.asm
  flash1.bin
```

to specify our particular example program, the instruction table TASM51.tab and binary output for-

Software

To load a program into the microcontroller on the 89S8252 Flash Board, you will need the Windows program **MicroFlash.exe**, which can be found on the *Elektor Electronics* website (www.elektor-electronics.co.uk) on the Free Downloads page, see the list for the December 2001 issue.

For the programming course, the TASM assembler is all you need to get started, but later on you will need the Rigel READ51 C compiler and the BASIC-52 Basic compiler. The TASM assembler is a popular program, which can be obtained together with TASMedit from the download list for this issue on the *Elektor Electronics* website. Please register the software and pay the programmer (T.N. Anderson) his well-earned fee. No payment is required for the C compiler, which Rigel make available free of charge for private and educational use. This compiler can be obtained from www.rigelcorp.com. BASIC-52 is a Basic interpreter created by Intel, which was mask-programmed in the program memory of an 80C52 microcontroller that received the designation 80C52-AH-BASIC. This microcontroller was used for nearly two decades by electronic engineers and programmers and became internationally famous, mainly as a result of articles in Elektor Electronics.. Several years ago, Intel ceased production of this IC, but they released the programming language as open source for general use. The language has been further developed and also adapted for use with other microcontrollers. Probably the most advanced version, V1.3, was presented in the February 2001 issue of *Elektor Electronics* and is available from Readers Services on diskette (order number **000121-11**).

The diskette for this course (Readers Services order number **010208-11**) contains the TASM assembler, TASMedit and the first sample programs, along with BASIC-52, MicrFlash.exe and a small Basic terminal emulator program with its own sample programs.

Number formats

The fact that we use a decimal number system is probably due to the fact that we happen to have ten fingers. The 'natural' number system for a computer is the binary system. The hexadecimal system represents a compromise, in which the range of numerals runs from 0 to 15, with the understanding that the numerals above 9 are represented by the letters A, B, C, D E and F.

Decimal	Hexadecimal	Binary
0	00h	00000000b
1	01h	00000001b
2	02h	00000010b
3	03h	00000011b
...	...	...
10	0Ah	00001010b
11	0Bh	00001011b
12	0Ch	00001100b
13	0Dh	00001101b
14	0Eh	00001110b
15	0Fh	00001111b
16	10h	00010000b
17	11h	00010001b
...	...	...
253	FDh	11111101b
254	FEh	11111110b
255	FFh	11111111b

In assembler programs, it is generally possible to choose which notation you want to use. If you are describing how an 8-bit port is being driven, binary notation is particularly clear. For example, the rightmost bit represents pin P1.0 and the leftmost bit represents pin P1.7. Here eight leads require eight bits, or one byte.

When TASM translates a program, you can specify the format in which the results are to be stored. In the binary format, only the bytes that represent the individual machine-language instructions are written. A text editor cannot make any sense of such a file.

The Intel hex format uses text lines containing hexadecimal numbers. In addition to the actual code, there is a start address and a checksum for each line. This format can also be viewed as text.

bahramelectronic@bahramelectronic.tk

http://bahramelectronic.tk

mat. Naturally, not everyone likes to work with command lines like this. Consequently, they won't be used at all in our course.

In general, we want to work only with Windows, but TASM is still a pure DOS program. For this reason, a Windows interface for the program, called TASMedit.exe, has been written. It includes its own editor and allows the user to immediately see the result of the translation, including possible error messages. The flash download tool for the *Elektor Electronics* Flash Board is also integrated into this program. No effort has been spared to make things as easy as possible for course participants!

The program has two text windows (see **Figure 3**). On the left there is the assembler source text editor. It can be used to manually enter a program or load a program from the hard disk. The TASM button starts the assembler in the background. The window on the right displays the assembler's list file along with any error messages that may be present. Download progress when the program is being sent to the microcontroller board is also shown in this window.

Clicking on the TASM button first generates a file called Work.asm, which contains the current content of the Editor window. This working text is then translated. TASM is called from the Windows interface using the command line

```
TASM -51 -b -work.asm work.bin
```

This means that the binary format is always used here. The result of the translation is stored in a file called Work.bin. This is the file that is read by the download module when the RUN button is actuated. The assembler also generates a file called Work.list containing the list file, which holds the translation in a readable form. The fact that the same file names are always used for the translation is an advantage for experimental work with the assembler, since the source text only has to be saved after the latest attempt has been successful. This means that we do not have a whole collection of garbage data on the hard disk from all the unsuccessful attempts, but instead only the intentionally saved source text and the work files for the most recent attempt. If you forget to save the latest version of the source text, or if the PC crashes while you are working, you can always use the Work.asm file to recover the fruits of your hard work.

When TASM is automatically started from the Windows interface, it appears in a DOS window (see **Figure 4**). This window must be closed before you can proceed. At first, it may be very enlightening to see how TASM goes about its work, but after the third time or so it will only annoy pampered Windows users. Consequently, from now on we would like to have the window be automatically closed. This is no problem, since Windows provides a solution. First click on the TASM.EXE file with the right mouse button and open the Properties menu. Under Properties / Program you will find the setting 'Close when done'. Enable this setting (see **Figure 5**). Windows then generates a link in the form of a file called TASM.PIF. From now on, the DOS window will automatically close after TASM has finished its job.

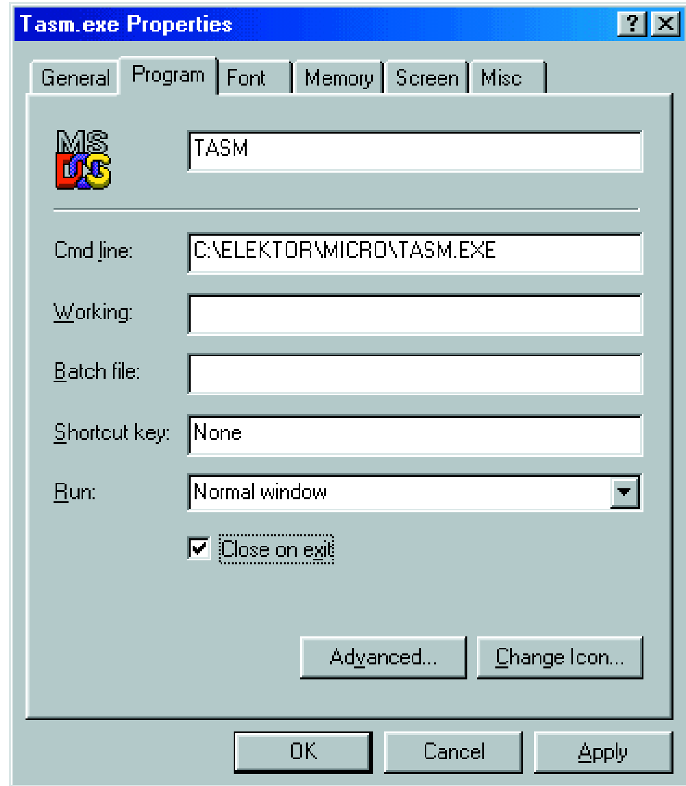

Figure 5. Setting the property 'close when done'.

Once a program has been successfully translated, the RUN button can be used to transfer it to the Flash Board system and start it. For this, you have to select a PC COM port and connect it to the board's programming connector (K2). If you have also looked after the most important prerequisite (applying the supply voltage to the Flash Board), you can then start to test the program. In the case of our first example, all you have to do is to observe the states of the Port 1 outputs in order to see whether the result is successful. Using a high-impedance meter, you should see almost exactly 5 V on port leads P1.0 through P1.3 and nearly 0 V on P1.4 through P1.7 (to be precise, around 30 mV flowing into ground).

In the ground state without any program running, or following a processor reset, all of the port leads take on the High state, with a voltage of 5 V on each pin. This can easily be checked using an oscilloscope or multimeter. The newly loaded and started program changes the states of four lines. P1.4 through P1.7 should now be Low, which means that they have a voltage of around 0 V, while P1.0 through P1.3 remain High. The program has transferred the value 15 (= 0Fh) to Port 1. This bit pattern can be seen on the port pins.

(010208-2)

This concludes our brief introduction to working with the assembler. In the next instalment of the course, we will discuss small sample programs that are primarily intended to help investigate the port properties of the microcontroller. We will look at inputs, outputs and achievable speeds.

bahramelectronic@bahramelectronic.tk